Algorithmique de base

I. Introduction

1. Algorithme

La notion d'algorithme et introduit par un mathématicien persan du 9ème siècle Al-Khawarizmi.

Algorithme c'est une méthode de calcul qui indique la démarche à suivre pour résoudre une série de problèmes équivalents en appliquant dans un ordre précis une suite finie de règles.

2. Algorithmique

L'algorithmique est la science des algorithmes.

L'algorithmique c'est de savoir comment lire, écrire, évaluer et optimiser des algorithmes.

Pourquoi apprendre l'algorithmique pour apprendre à programmer ? En quoi a-t-on besoin d'un langage spécial, distinct des langages de programmation compréhensibles par les ordinateurs ?

Parce que l'algorithmique exprime les instructions résolvant un problème donné indépendamment des particularités de tel ou tel langage.

Apprendre l'algorithmique, c'est apprendre à manier la structure logique d'un programme informatique.

3. Programme

Un programme est donc une suite d'instructions exécutées par la machine. La machine a son propre langage appelé langage machine.

Un programme est l'expression d'un algorithme par une machine donnée dans un langage de programmation donné en utilisant le répertoire d'actions (opérations, instructions) et les règles de composition propres à cette machine et à ce langage donnés.

Un programme est un assemblage et un enchaînement d'instructions élémentaires écrit dans un langage de programmation, et exécuté par un ordinateur afin de traiter les données d'un problème et renvoyer un ou plusieurs résultats.

4. Méthodologie

Pour résoudre un problème, il est vivement conseillé de réfléchir d'abord à l'algorithme avant de programmer.

La résolution d'un problème est caractérisée par 4 étapes :

- Comprendre la nature du problème posé
- Préciser les données fournies (Entrée)

- Préciser les résultats que l'on désire obtenir (Sortie)
- Déterminer le processus de transformation des données en résultats.

II. Notions de Base

Les algorithmes sont rédigés dans un langage à mi-chemin entre le français et les langages de programmation, dit **pseudo-code**.

En programmation, le **pseudo-code** est une façon de décrire un algorithme sans référence à un langage de programmation particulier. L'écriture en **pseudo-code** permet souvent de bien prendre toute la mesure de la difficulté de l'implémentation de l'algorithme, et de développer une démarche structurée dans la construction de celui-ci.

Ce mode de représentation consiste à exprimer en langage naturel, mais selon une disposition particulière et des mots choisis.

Ce **pseudo-code** est susceptible de varier légèrement d'un livre (ou d'un enseignant) à un autre. C'est bien normal : le **pseudo-code**, encore une fois, est purement conventionnel.

5. Les variables

Dans un programme informatique, on va avoir en permanence besoin de stocker provisoirement des valeurs. Il peut s'agir de données issues du disque dur, fournies par l'utilisateur (frappées au clavier. Il peut aussi s'agir de résultats obtenus par le programme, intermédiaires ou définitifs. Ces données peuvent être de plusieurs types : elles peuvent être des nombres, du texte, etc.

Pour pouvoir accéder aux données, le programme fait usage d'un grand nombre de **variables** de différents types.

5.1. Déclaration des variables

Une variable peut être représentée par une case mémoire, qui contient la valeur d'une donnée.

Chaque variable possède un nom unique appelé identificateur par lequel onpeut accéder à son contenu.

Par exemple, on peut avoir en mémoire une variable **prix** et une variable **quantité**.

Une variable possède 3 attributs :

- Une valeur
- Un nom(invariable) qui sert à la désigner
- Un type(invariable) qui décrit l'utilisation possible de la variable

⇒ ***Une valeur***

La valeur d'une variable(contenu) peut varier au cours du programme. L'ancienne valeur est tout simplement écrasée et remplacée par la nouvelle.

⇒ ***Nom de la variable***

C'est une suite de lettres et de chiffres commençant nécessairement par une lettre, le nombre maximal de caractères imposé varie selon les langages

La lisibilité des programmes dépend de l'habilité à choisir des noms représentatifs.

On évitera tout caractère accentué dans les noms de variable.

⇒ ***Type de la variable***

Le type de la variable définit la nature des informations qui seront représentées dans la variable

Propriété *: Une variable doit être déclaré avant son utilisation*

- **Entier** : il s'agit des variables destinées à contenir un nombre entier positif ou négatif.
- **Réel :** il s'agit des variables numériques qui ne sont pas des entiers, c'est à dire qui comportent des décimales
- **Caractère :** Les variables de type caractère contiennent des caractères alphabétiques ou numériques seul(ex: 'c')
- **Booléen :** Les variables qui prennent les valeurs (vrai ou faux) ou les valeurs (oui ou non).
- **Chaine:** représentant un texte, contenant un ou plusieurs caractères(ex: ''Bonjour tout le monde'')

Tous les traducteurs de langages prennent en compte cette notion de type par des instructions de déclaration de type ;

```
Variable    Moyenne :  réel;
Variable    NbreEtudiant :  entier;
Variables     c1, lettre, z :  caractères;
Variables nom, prenom : chaine ;
```

⇒ ***Constante :***

Une constante est un objet de valeur invariable.

Exemple :

```
Constante pi <- 3.14 : réel;
Constante max <- 100 : entier ;
```

5.2. Ou sont déclarées les variables ?

On déclare les variables dans le bloc **déclaration**

```
Algorithme   Nom_de_l_algorithme :
        Déclaration;
Début
        Actions;
Fin
```

⇒ ***Exemple***

```
Algorithme   Nom_de_l_algorithme :
  Variables a,b : entiers ;
  Variable f : réel ;
  Constante pi <- 3.14 : réel ;
Début
   Actions;
Fin
```

6. Instruction d'affectation

L'instruction d'affectation permet de manipuler les valeurs des variables. Son rôle consiste à placer une valeur dans une variable.

⇒ ***Notation :***

- Nom_variable <- valeur;

⇒ ***Exemple :***

1. Affecter une valeur à une variable X <-5;
   - On charge la variable X avec la valeur 5
2. Affecter le contenu d'une variable à une autre variable X <- Y;
   - On charge X avec le contenu de Y , X et Y doivent être de même type
3. Affecter une formule à une variable X <- X + 2 * Y;
   - On charge la variable X par la valeur du résultat de la formule et on écrase sa valeur initiale.

7. Instructions d'Entrée/Sortie

Un programme est amené à :

- Prendre des données par le périphérique(clavier) : rôle de l'instruction de lecture
- Communiquer des résultats par l'intermédiaire du périphérique(écran) : rôle de l'instruction de l'écriture

7.1. Instruction de lecture :

Fournir des données au programme

- ***Syntaxe: Lire(variable)***
- ***Exemple: Lire(X)*** on saisie une valeur pour la stocker après dans la variable X

7.2. Instruction d'écriture

Fournir des résultats directement compréhensibles

- ***Syntaxe:*** Ecrire(variable), Ecrire("chaine de caractères")
- ***Exemple:*** Ecrire(X), Ecrire("Bonjour") , Ecrire ("a=",a), Ecrire ("a=",a,"b=",b)

7.3. Exemple

Ecrire un algorithme qui permet de calculer la somme de 2 valeurs entiers ;

- ***Corrigé :***
  - On a besoin de stocker les deux variables d'entrée dans la mémoire donc déclarer deux variables de type entier ; (données d'entrée)
  - L'algorithme doit fournir à l'utilisateur le résultat de calcul pour cela il doit stocker le résultat dans une autre variable(données de sortie)
  - Les valeurs des variables d'entrée sont **inconnues** donc l'utilisateur doit fournir des valeurs au programme (opération de lecture)
  - Pour une interaction entre le programme et l'utilisateur, le programme doit afficher des messages pour l'utilisateur pour lui demander de saisir les valeurs des données d'entrée.

```
Algorithme   somme :
   Variables a,b : entiers ;// variables d'entrée
   Variable s : entier;// variable de sortie
Début
   Ecrire ("Saisir la première valeur : ") ;
   Lire(a) ;
   Ecrire ("Saisir la deuxième valeur : ") ;
   Lire(b) ;
   S <- a+b ;
   Ecrire ("la somme = ",s) ;// Ecrire ("la somme de ",a," et ",b," =",s) ;
Fin
```

III. Les structures de contrôle conditionnelles

Un programme est donc une suite d'instructions exécutées par la machine. Ces instructions peuvent s'exécuter dans certains cas et pas dans d'autres, on parle alors de structure alternative ;

1. L'alternative SI-ALORS-SINON

Une alternative s'exprime par si … Sinon…

1.1. Alternative simple

⇒ ***Syntaxe1:***

```
Si<conditions>   alors
          <Instruction1>
   …
          <Instruction N>
Fin Si
```

⇒ ***Exemple***

```
Si (a<0) alors
      Ecrire( "la valeur de a et négatif");
Fin Si
```

⇒ ***Syntaxe 2:***

```
Si <expression booléenne>  alors
          <Instruction1>
          …
          <Instruction N>
Sinon
          <Instruction1>
          …
          <Instruction N>
Fin si
```

⇒ ***Exemple :***

```
Si (a<b) alors
      max <- b
Sinon
      max <-  a
Fin si
```

2. Qu'est ce qu'une condition ?

Une condition est composée de trois éléments : une valeur, un **opérateur de comparaison,** une autre valeur

Les valeurs peuvent être a priori de n'importe quel type (numériques, caractères…). Mais si l'on veut que la comparaison ait un sens, il faut que les deux valeurs de la comparaison soient du même type !

Type	Exemples	Opérations possibles	symboles
Réel	-15.69, 0.49	Addition	+
		Soustraction	-
		Multiplication	*
		Division	/
		Exposant	^
		Pourcentage	%
		comparaisons	<,<=,>,>=,=,…
Entier	-10, 4, 768	Addition	+
		Soustraction	-
		Multiplication	*
		Division	DIV
		Modulo	MOD
		Exposant	^
		Pourcentage	%
caractère	'B', '\n'	comparaisons	<,<=,>,>=,=,…
Booléen	Vrai, Faux	Comparaison	<,<=,>,>=,=,…
		Négation	NON
		Conjonction	ET
		disjonction	OU

⇒ ***Exercice 1:***

Ecrire un algorithme qui permet d'afficher le message « non admis » si la note de l'étudiant et inférieure a 10 et admis dans le cas ou la note supérieure à 10

- ***Corrigé :***

```
Algorithme admission :
  Variable note : réel ;
Début
  Ecrire ("Saisir une note : ") ;
  Lire(note) ;
  SI note < 10 ALORS
    Ecrire("Non admis")
  SINON
    Ecrire("Admis")
  FIN SI
Fin
```

3. Conditions composées :

Certains problèmes exigent parfois de formuler des conditions composées liées entre eux par les opérateurs logiques suivants : **ET, OU, NON, et XOR**.

Tables de vérité (C1 et C2 représentent deux conditions, et on envisage à chaque fois les quatre cas possibles) :

C1 et C2	C2 Vrai	C2 Faux
C1 Vrai	Vrai	Faux
C1 Faux	Faux	Faux

C1 ou C2	C2 Vrai	C2 Faux
C1 Vrai	Vrai	Vrai
C1 Faux	Vrai	Faux

Non C1	
C1 Vrai	Faux
C1 Faux	Vrai

4. *Tests imbriqués :*

Dans les cas précédents on a seulement deux situation possibles (jour et nuit), (admin, non admis) mais s'il y'a plusieurs situations par exemple dans la mention (non admin, passable, assez bien, bien, très bien) une structure alternative simple ne suffit pas, donc pour traiter ce cas on doit passer par plusieurs structures alternatives on parle (test imbriqué)

⇒ ***Syntaxe :***

```
Si <condition 1>  alors
        <Instruction1>
Sinon Si <condition 2>  alors
        <Instruction1>
...
Sinon
        <Instruction1>
        <Instruction N>
Fin Si
```

⇒ ***Exercice 2 :***

Ecrire un algorithme qui permet de tester le signe d'un nombre ;

- ***Corrigé :***

```
Algorithme  signe_nombre  :
  Variables nb: entier ;
Début
  Ecrire ("Saisir un nombre : ") ;
  Lire(nb) ;
  SI nb<0 ALORS
     Ecrire("le nombre est négatif")
  SINON SI nb > 0 ALORS
     Ecrire("le nombre est positif")
  SINON
     Ecrire("le nombre est nul")
  FIN SI
Fin
```

⇒ ***Exercice 3***

Écrire un algorithme qui demande l'âge d'un enfant à l'utilisateur. Ensuite, il l'informe de sa catégorie :

- « Poussin » de 6 à 7 ans
- « Pupille » de 8 à 9 ans
- « Minime » de 10 à 11 ans
- « Cadet » après 12 ans

- ***Corrigé :***

Solution 1

```
Algorithme   categorie_enfant  :
  Variable age : entier ;
Début
  Ecrire ("Saisir l'age: ") ;
  Lire(age) ;
  SI age < 6 ALORS
    Ecrire("Saisir un age supérieur ou égal 6")
  SINON SI age <=7 ALORS
    Ecrire("Poussin")
  SINON SI age <=9 ALORS
    Ecrire("Pupille")
  SINON SI age <=11 ALORS
    Ecrire("Minime")
  SINON
    Ecrire("Cadet")
  FIN SI
Fin
```

Solution 2

```
Algorithme   categorie_enfant  :
  Variable age : entier ;
Début
  Ecrire ("Saisir l'age: ") ;
  Lire(age) ;
  SI age < 6 ALORS
    Ecrire("Saisir un age supérieur ou égal 6")
  SINON SI age >=6 ET age <=7 ALORS
    Ecrire("Poussin")
  SINON SI age >=8 ET age <=9 ALORS
    Ecrire("Pupille")
  SINON SI age >=10 ET age <=11 ALORS
```

```
    Ecrire("Minime")
  SINON
    Ecrire("Cadet")
  FIN SI
Fin
```

⇒ ***Exercice 4***

Écrire un algorithme qui à partir d'un nombre compris entre 1 et 7 affiche le jour correspondant ?

- ***Corrigé :***

```
Algorithme  jour_semaine  :
  Variable jour: entier ;
Début
  Ecrire ("Saisir un jour : ") ;
  Lire(nb) ;
  SI jour=1 ALORS
    Ecrire("Lundi")
  SINON SI jour=2 ALORS
    Ecrire("Mardi")
  SINON SI jour=3 ALORS
    Ecrire("Mercredi")
  SINON SI jour=4 ALORS
    Ecrire("Jeudi")
  SINON SI jour=5 ALORS
    Ecrire("Vendredi")
  SINON SI jour=6 ALORS
    Ecrire("Samedi")
  SINON SI jour=7 ALORS
    Ecrire("Dimanche")
  SINON
    Ecrire("jour invalide")
  FIN SI
Fin
```

5. *Structure à choix multiples SELON*

La structure **SELON** permet d'effectuer tel ou tel traitement en fonction de la valeur du sélecteur (variable ou donnée) .

⇒ ***Syntaxe:***

```
Selon sélecteur faire
   valeur 1 : <Traitement1>
   valeur 2 : <Traitement2>
    … …
    … …
   valeur N : <Traitement N>
   SINON
        <Traitement R>
Fin selon
```

⇒ ***Explication :***

- Si le sélecteur vaut valeur 1, alors on exécute le traitement1 et on quitte la structure Selon.
- Si le sélecteur et différent de valeur1, on évalue la valeur2...et ainsi de suite.
- Si **aucune n'est vraie** on effectue l'**action sinon** (au cas où l'action sinon n'existe pas alors aucune action n'est exécutée !).

⇒ ***Exemple:***

Écrire un algorithme qui à partir d'un nombre compris entre 1 et 7 affiche le jour correspondant ?

```
Algorithme  jours_semaine  :
  Variable jour : entier ;
Début
  Ecrire ("Saisir une valeur : ") ;
  Lire(jour) ;
  SELON jour faire
    1 : Ecrire("Lundi")
    2 : Ecrire("Mardi")
    3 : Ecrire("Mercredi")
    4 : Ecrire("Jeudi")
    5 : Ecrire("Vendredi")
    6 : Ecrire("Samedi")
    7 : Ecrire("Dianche")
    SINON : Ecrire("Jour invalide")
   Fin SELON
Fin
```

IV. Les structures répétitives

Un programme est donc une suite d'instructions exécutées par la machine. Ces instructions peuvent se répéter plusieurs fois, on parle alors de structure répétitive.

On identifie en règle générale 3 types de répétitive :

- Tant Que
- Répéter Jusqu'à
- Pour

⇒ ***Exemple :***

La routine journalière d'un employé est :

```
Ouvrir guichet
Appeler premier client
Tant Que " client dans file  d'attente " et " pas fin de journée "
Traiter client
        Appeler client suivant
Fin Tant Que
```

Les deux actions "Traiter client" et "Appeler client suivant" vont se répéter tant que la condition située derrière l'instruction "Tant que" est vérifiée.

1. *La structure Tant que*

La structure Tant que permet de répéter l'action tant que la condition est réalisée. Le nombre de répétition n'est pas connu à l'avance.

⇒ ***Syntaxe :***

```
Tant que <expression  logique> Faire
   <Traitements>
Fin Tant que
```

Le principe est simple : l'algorithme arrive sur la ligne du Tant Que. Il examine l'expression logique (qui, je le rappelle, peut être une variable booléenne ou, plus fréquemment, une condition). Si cette valeur est VRAI, l'algorithme exécute les instructions qui suivent, jusqu'à ce qu'il rencontre la ligne Fin Tant Que. Il retourne ensuite sur la ligne du Tant Que, procède au même examen, et ainsi de suite. Le cycle ne s'arrête que lorsque le booléen prend la valeur FAUX.

⇒ ***Exemple :***

Ecrire un algorithme qui demande à l'utilisateur un nombre compris entre 1 et 3 jusqu'à ce que la réponse convienne.

```
Algorithme nombre
  Variable nombre : entier
Début
  Ecrire("Saisir une valeur");
  Lire(nombre)
  Tant que nombre >3 OU nombre < 1 Faire
      Ecrire("Saisir une valeur");
```

```
    Lire(nombre)
  Fin Tant que
Fin
```

2. *Répéter..Jusqu'à :*

La condition d'arrêt est placée à la fin

⇒ ***Syntaxe :***

```
Répéter
   <Traitement>
Jusqu' à (condition d'arrêt)
```

Cet ordre d'itération permet de répéter le <Traitement> une ou plusieurs fois et de s'arrêter sur une condition. En effet, lorsque la condition est vérifiée, la boucle s'arrête, si non elle ré-exécute le <Traitement>.

⇒ ***Remarques***

- Dans cette boucle, le traitement est exécuté au moins une fois avant l'évaluation de la condition d'arrêt.
- Il doit y avoir une action dans le <Traitement> qui modifie la valeur de la condition.

⇒ ***Exemple :***

Ecrire un algorithme qui demande à l'utilisateur un nombre compris entre 1 et 3 jusqu'à ce que la réponse convienne.

```
Algorithme nombre
  Variable nombre : entier
Début
   Répéter
     Ecrire("Saisir une valeur");
     Lire(nombre)
   jusqu'à  nombre >= 1 ET nombre <= 3
Fin
```

3. *La structure Pour*

Il arrive très souvent qu'on ait besoin d'effectuer un nombre déterminé de passages. Or, a priori, notre structure TantQue ne sait pas à l'avance combien de tours de boucle elle va effectuer ;

Les répétitives où le nombre d'itération est fixée une fois pour toute.

⇒ ***Syntaxe***

```
Pour Compteur de valeur_initiale à valeur_finale PAS le pas Faire
   Actions à répéter
Fin Pour
```

La progression du compteur est laissée à votre libre disposition. Dans la plupart des cas, on a besoin d'une variable qui augmente de 1 à chaque tour de boucle. On ne précise alors rien à l'instruction « Pour » ; celle-ci, par défaut, comprend qu'il va falloir procéder à cette incrémentation de 1 à chaque passage, en commençant par la première valeur et en terminant par la deuxième.

Si vous souhaitez une progression plus spéciale, de 2 en 2, ou de 3 en 3, ou en arrière, de –1 en –1, ou de –10 en –10, il faut préciser à votre instruction « Pour » en lui rajoutant le mot « **Pas** » et la valeur de ce pas.

Quand on met un pas négatif dans une boucle, la valeur initiale du compteur doit être supérieure à sa valeur finale si l'on veut que la boucle tourne !

⇒ ***Exemple :***

Écrire un algorithme pour calculer et afficher la somme de 100 premiers nombres entiers ?

```
Algorithme somme
Variables compteur, somme : entiers
Début
  somme <- 0 ;
  Pour compteur de 1 à 100 Faire
      somme <- somme + compteur;
  Fin Pour
  Ecrire("la somme =",somme)
Fin
```

Les structures **TantQue** sont employées dans les situations où l'on doit procéder à un traitement sur les éléments d'un ensemble dont on ne connaît pas d'avance la quantité, comme par exemple

- le contrôle d'une saisie.
- la gestion des tours d'un jeu (tant que la partie n'est pas finie, on recommence).

Les structures **Pour** sont employées dans les situations où l'on doit procéder à un traitement sur les éléments d'un ensemble dont on connaît d'avance la quantité.

V. Procédures et fonctions

1. Introduction

Lorsque l'on progresse dans la conception d'un algorithme, ce dernier peut prendre une taille et une complexité croissante. De même des séquences d'instructions peuvent se répéter à plusieurs endroits.

Un algorithme écrit d'un seul tenant devient difficile à comprendre et à gérer dès qu'il dépasse deux pages. La solution consiste alors à découper l'algorithme en plusieurs parties plus petites. Ces parties sont appelées des sous-algorithmes.

Le sous-algorithme est écrit séparément du corps de l'algorithme principal et sera appelé par celui-ci quand ceci sera nécessaire.

Il existe deux sortes de sous-algorithmes : les procédures et les fonctions.

2. Les procédures

Une procédure est une série d'instructions regroupés sous un nom, qui permet d'effectuer des actions par un simple appel de la procédure dans un algorithme ou dans un autre sous-algorithme.

Une procédure renvoie plusieurs valeurs (par une) ou aucune valeur.

2.1. Déclaration d'une procédure

⇒ ***Syntaxe :***

```
Procédure nom_proc(liste de paramètres)
Variables identificateurs : type
Début
        Instruction(s)
FinRoc
```

Après le nom de la procédure, il faut donner la liste des paramètres (s'il y en a) avec leur type respectif. Ces paramètres sont appelés paramètres formels. Leur valeur n'est pas connue lors de la création de la procédure.

⇒ ***Exemple :***

Ecrire une procédure qui affiche à l'écran une ligne de 15 étoiles puis passe à la ligne suivante.

- ***Solution :***

```
Procédure Etoile()
Variables i : entier
Début
      Pour i Allant de 1 à 15 faire
         Afficher("*")
      FinPour
      //\n : retour à la ligne
      Afficher("\n")
FinProc
```

2.2. L'appel d'une procédure

Pour déclencher l'exécution d'une procédure dans un programme, il suffit de l'appeler.

L'appel de procédure s'écrit en mettant le nom de la procédure, puis la liste des paramètres, séparés par des virgules.

A l'appel d'une procédure, le programme interrompt son déroulement normal, exécute les instructions de la procédure, puis retourne au programme appelant et exécute l'instruction suivante.

⇒ ***Syntaxe :***

```
Nom_proc(liste de paramètres)
```

Les paramètres utilisés lors de l'appel d'une procédure sont appelés paramètres effectifs. Ces paramètres donneront leurs valeurs aux paramètres formels.

⇒ ***Exemple :***

En utilisant la procédure Etoiles déclarée dans l'exemple précédent, écrire un algorithme permettant de dessiner un carré d'étoiles de 15 lignes et de 15 colonnes.

- ***Solution :***

```
Algorithme carré_étoiles
Variables j : entier

//Déclaration de la procédure Etoiles()
Procédure Etoile()
Variables i : entier
Début
  Pour i Allant de 1 à 15 Faire
     Afficher("*")
  FinPour
  Afficher("/n")
FinProc

//Algorithme principal (Partie principale)
Début
   Pour j Allant de 1 à 15 Faire
        //Appel de la procédure Etoiles
        Etoile()
   FinPour
Fin
```

⇒ ***Remarque 1 :***

Pour exécuter un algorithme qui contient des procédures et des fonctions, il faut commencer l'exécution à partir de la partie principale (algorithme principal)

⇒ ***Remarque 2 :***

Lors de la conception d'un algorithme deux aspects apparaissent :

La définition (déclaration) de la procédure ou fonction.

L'appel de la procédure ou fonction au sein de l'algorithme principal.

2.3. Passage de paramètres

Les échanges d'informations entre une procédures et le sous algorithme appelant se font par l'intermédiaire de paramètres.

Il existe deux principaux types de passages de paramètres qui permettent des usages différents :

2.4. Passage par valeur :

Dans ce type de passage, le paramètre formel reçoit uniquement une copie de la valeur du paramètre effectif. La valeur du paramètre effectifs ne sera jamais modifiée.

⇒ ***Exemple :***

Soit l'algorithme suivant :

```
Algorithme Passage_par_valeur
Variables N : entier

//Déclaration de la procédure P1
Procédure P1(A : entier)
Début
   A ← A * 2
   Afficher(A)
FinProc

//Algorithme principal
Début
  N ← 5
  P1(N)
  Afficher(N)
Fin
```

Cet algorithme définit une procédure P1 pour laquelle on utilise le passage de paramètres par valeur.

Lors de l'appel de la procédure, la valeur du paramètre effectif N est recopiée dans le paramètres formel A. La procédure effectue alors le traitement et affiche la valeur de la variable A, dans ce cas 10.

Après l'appel de la procédure, l'algorithme affiche la valeur de la variable N dans ce cas 5.

La procédure ne modifie pas le paramètre qui est passé par valeur.

2.5. Passage par référence ou par adresse :

Dans ce type de passage, la procédure ***utilise l'adresse*** du paramètre effectif. Lorsqu'on utilise l'adresse du paramètre, ***on accède directement à son contenu***. La valeur de la variable effectif sera donc modifiée.

Les paramètres passés par adresse sont précédés du mot clé ***Var***.

⇒ ***Exemple :***

Reprenons l'exemple précédent :

```
Algorithme Passage_par_référence
Variables N : entier

//Déclaration de la procédure P1
Procédure P1 (Var A : entier)
Début
   A ← A * 2
   Afficher(A)
FinProc

//Algorithme Principal
Début
    N ← 5
    P1(N)
    Afficher(N)
Fin
```

A l'exécution de la procédure, l'instruction ***Afficher(A)*** permet d'afficher à l'écran 10. Au retour dans l'algorithme principal, l'instruction ***Afficher(N)*** affiche également 10.

Dans cet algorithme le paramètre passé correspond à la référence (adresse) de la variable N. Elle est donc modifiée par l'instruction :

```
A ← A *2
```

⇒ ***Remarque :***

Lorsqu'il y a plusieurs paramètres dans la définition d'une procédure, il faut absolument qu'il y en ait le même nombre à l'appel et que l'ordre soit respecté.

3. Les fonctions

Les fonctions sont des sous algorithmes admettant des paramètres et retournant un seul résultat (une seule valeur) de type simple qui peut apparaître dans une expression, dans une comparaison, à la droite d'une affectation, etc.

3.1. Déclaration d'une fonction

⇒ ***Syntaxe :***

```
Fonction nom_Fonct (liste de paramètres) : type
Variables identificateur : type
Début
    Instruction(s)
    Retourner Expression
Fin
```

La syntaxe de la déclaration d'une fonction est assez proche de celle d'une procédure à laquelle on ajoute un type qui représente le type de la valeur retournée par la fonction et une instruction Retourner Expression. Cette dernière instruction renvoie au programme appelant le résultat de l'expression placée à la suite du mot clé Retourner.

⇒ ***Note :***

Les paramètres sont facultatifs, mais s'il n'y pas de paramètres, les parenthèses doivent rester présentes.

⇒ ***Exemple :***

Définir une fonction qui renvoie le plus grand de deux nombres différents.

- ***Solution :***

```
//Déclaration de la fonction Max
Fonction Max(X: réel, Y:réel) : réel
Début
    Si  X > Y Alors
        Retouner X
    Sinon
        Retouner Y
    FinSi
FinFonction
```

3.2. L'appel d'une fonction

Pour exécuter une fonction, il suffit de faire appel à elle en écrivant son nom suivie des paramètres effectifs. C'est la même syntaxe qu'une procédure.

A la différence d'une procédure, la fonction retourne une valeur. L'appel d'une fonction pourra donc être utilisé dans une instruction (affichage, affectation, ...) qui utilise sa valeur.

⇒ ***Syntaxe***

```
    Nom_Fonc(list de paramètres)
```

⇒ ***Exemple :***

Ecrire un algorithme appelant, utilisant la fonction Max de l'exemple précédent.

- ***Solution :***

```
Algorithme Appel_fonction_Max
Variables A, B, M : réel
//Déclaration de la fonction Max
Fonction Max(X: réel, Y: réel) : réel
Début
    Si X > Y Alors
        Retourner X
    Sinon
        Retourner Y
    FinSi
```

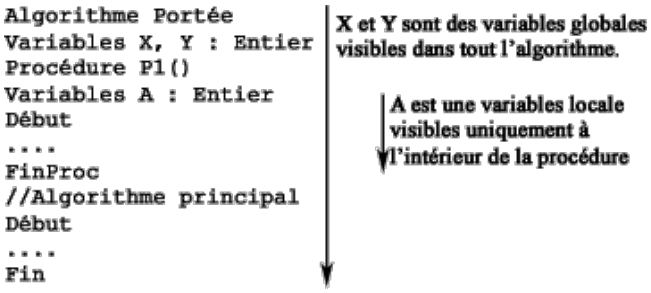
```
FinFonction

//Algorithme principal
Début
    Afficher("Donnez la valeur de A :")
    Saisir(A)
    Afficher("Donnez la valeur de B :")
    Saisir(B)
    //Appel de la fonction Max
    M ← Max(A,B)
    Afficher("Le plus grand de ces deux nombres est : ", M)
Fin
```

4. *Portée des variables*

La portée d'une variable désigne le domaine de visibilité de cette variable. Une variable peut être déclarée dans deux emplacements distincts.

Une variable déclarée dans la partie déclaration de l'algorithme principale est appelée variable globale. Elle est accessible de n'importe où dans l'algorithme, même depuis les procédures et les fonctions. Elle existe pendant toute la durée de vie du programme.

Une variable déclarée à l'intérieur d'une procédure (ou une fonction) est dite locale. Elle n'est accessible qu'à la procédure au sein de laquelle elle définie, les autres procédures n'y ont pas accès. La durée de vie d'une variable locale est limitée à la durée d'exécution de la procédure.



⇒ ***Remarque :***

Les variables globales sont à éviter pour la maintenance des programmes.

5. *La récursivité*

Une procédure (ou une fonction) est dite récursive si elle s'appelle elle-même.

⇒ ***Exemple :***

Ecrire une fonction récursive permettant de calculer la factorielle d'un entier positif.

- ***Solution :***

n != n * (n-1) ! : la factorielle de n est n fois la factorielle de n-1 :

```
//Déclaration de la fonction Factorielle (Fact)
Fonction Fact(n : entier) : entier
Début
    Si n > 1 Alors
        Retourner (fact(n-1)*n)
    Sinon
        Retourner 1
    FinSi
FinFonction
```

Dans cet exemple, la fonction renvoie 1 si la valeur demandée est inférieur à 1, sinon elle fait appel à elle même avec un paramètre inférieur de 1 par rapport au précédent. Les valeurs de ces paramètres vont en décroissant et atteindront à un moment la valeur une (1). Dans ce cas, il n'y a pas d'appel récursif et donc nous sortons de la fonction.

⇒ ***Note :***

Toute procédure ou fonction récursive comporte une instruction (ou un bloc d'instructions) nommée "point terminal" permettant de sortir de la procédure ou de la fonction.

Le "point terminal" dans la fonction récursive Fact est : retourner 1.

1. *Avantages des procédures et fonctions*

Les procédures ou fonctions permettant de ne pas répéter plusieurs fois une même séquence d'instructions au sein du programme (algorithme).

La mise au point du programme est plus rapide en utilisant des procédures et des fonctions. En effet, elle peut être réalisée en dehors du contexte du programme.

Une procédure peut être intégrée à un autre programme, ou elle pourra être rangée dans une bibliothèque d'outils ou encore utilisée par n'importe quel programme.